Views, Objects, and Databases

Gio Wiederhold, Stanford University

Objects provide the useful abstraction in programming languages; views provide a similar abstraction in databases. Since databases provide for persistent and shared data storage, view concepts will avoid problems occurring when persistent objects are to be shared.

The intention of this article is twofold. First of all, I show intrinsic differences in the underlying concepts of access to persistent storage in databases and current extensions of object-oriented programming systems intended to incorporate persistence.[1,2] Since the objectives of both paradigms are similar, I develop a connection between *object* concepts in programming languages and *view* concepts in database systems.

Secondly, I propose an architecture that exploits the commonality of objects and views, and indicate research and development directions needed to make the bridge viable. Such an architecture seems to be especially suitable for computer-aided design tasks.

Engineering information systems, or EISs, and systems for similar applications must provide suitable abstractions of real-world objects for their users and support long-lived transactions.[3] The complexity of the design process and the number of specialized individuals needed to bring major projects to completion are driving the search for more systematized solutions than those provided by the file mechanisms now in use in operational precursors of EISs. The issues being raised here pertain to systems with large quantities of data and long lifetimes. Design tasks that can be handled by single individuals are not likely to benefit from EIS technology.

Databases and views

Database concepts provide independence of storage and user data formats. The schema describes the form of the database contents, so that a variety of users can satisfy their data requirements by querying the shared database using nonprocedural declarations of the form:

SELECT what-I-want WHERE some-set-of-conditions-is-true

A database administrator integrates the requirements of diverse users. The shared database can be changed as long as the schema is changed to reflect such changes. Concepts of database normalization help avoid redundancy of storage and anomalies that are associated with updates of redundantly stored information.

The principal formal database mechanism to obtain selected data for an application is a view specification. A view on a database consists of a query that defines a suitably limited amount of data. A database administrator is expected to use predefined views as a management tool. Having predefined views simplifies the user's access and can also restrict the user from information that is to be protected. We are interested here only in the first objective, and not in the protection aspects associated with views.

Views have seen formal treatment in relational databases, although subset definitions without structural transformations are common in other commercial database systems. I will hence describe views from a relational point of view, without implying that a relational implementation is best for the underlying EIS databases.

A view is defined in a relational model as a query over the base relations, and perhaps also over other views. Current implementations do not materialize views, but transform user operations on views into operations over the base data. The final result is obtained by interpreting the combination of view definition and user query, using the description of the database stored in the database schema.

The view, like any relational query result, is a relation. However, even when the base database is fully normalized, say to Boyce-Codd normal form, the view relation is, in general, only in first normal form. Views are in that sense already closer to objects: related data has been brought together.

The issue that views present data not in second or third normal form seems ignored in database research, except for the update complications that result.[4] I have no evidence that the unnormalized views are uncomfortable to a user expecting a normalized relational representation. Acceptance of unnormalized views can be taken as a partial validation of the acceptability of structures more suitable to represent objects than normalized tables. Some current research is addressing unnormalized relations independent from the view aspect.

Objects

Object-oriented programming languages help to manage related data having a complex structure by combining them into objects. An object instance is a collection of data elements and operations that is considered an entity. Objects are typed, and the format and operations of an object instance are inherited from the object prototype.

The prototype description for the object type is predefined and the object instances are instantiated as needed for the particular problem. The object prototype then provides a meta description, similar to a schema provided for a database. That description is, however, fully accessible to the programmer. Internally, an object can have an arbitrary structure, and no user-visible join operations are required to bring data elements of one object instance together.

The object concept covers a range of approaches. One measure is the extent to which messages to external operation interfaces are used to provide access and manipulation functions. Objects may be active, as in the Actors paradigm, or passive, as in CLU, or somewhere in between, as in Simula or Smalltalk in terms of initiating actions.

The use of objects permits the programmer to manipulate data at a higher level of abstraction. Convincing arguments have been made for the use of object-oriented languages and some impressive demonstrations exist. Especially attractive is the display capability associated with objects. Object concepts can of course be implemented using nonspecialized languages, for instance in Lisp or Prolog. The tasks in EIS seem to match object-oriented concepts well and many experiments have been conducted.

Objects and databases

Let us assume a database is used for persistent storage of shared objects. A database query can obtain the information for an object instance, and an object-oriented programming language can instantiate that object. An interface is needed, since neither can perform the task independently. A view can define the information required for a collection of objects, but the data will not be arranged as expected for a collection of objects. Linkage to the object prototype and its operations is performed in the programming system. The program queries the database nonprocedurally to remain unaffected by database changes made to satisfy other users.

The query needed to instantiate an object may seem quite complex to a programmer. A relation is a set of tuples, and, from an idealized point of view, each tuple provides the data for some object. However, normalization often requires that information concerning one object be distributed over multiple relations, and brought together as needed by join operations. The base relations must contain all the data required to construct the view tuples; the composition knowledge is encoded into the Select expressions used to construct the views. An ideal composition of an object should allow its data to be managed as a unit, but unless non-first-normal form relations—supporting repeating groups—are supported for views, multiple tuples are still required to represent one entity in a relational view.

Hence storage of objects is not easy in databases, as indicated by the extensions proposed to Ingres for such tasks.[5] A further complexity is that objects themselves may be composed from more primitive objects. In hierarchical databases records may be assembled from lower level tuples, but in relational databases the programmer has to provide join specifications externally to assemble more comprehensive relations from basic relations.

There is some hope that the formal techniques being developed for databases can permit the management of the information required to manage objects. Performance remains a bottleneck, and I will consider this issue later with my proposal. The database community has to be careful not to promise too much too soon to the object-oriented folk.

Sharing information in objects

More serious are the problems I see in the management of shared information within the object-oriented paradigm. I consider two problems:

(1) The growth of objects that contain information for multiple design tasks.

(2) The conflict when object configurations differ for successive design tasks.

Let us first consider the simpler case, where multiple users deal with the same configuration of objects. I will draw my examples from EIS, and consider that the objects are components of a circuit.

In using an EIS the level of abstraction for the objects changes during the process of design. First the objects are simple logic elements and the process of design refines these objects to circuit components and, eventually, to simple geometric elements projected from each layer of the chip. The final objects will carry many data elements not needed at the higher levels. The sketch in Figure 1 symbolizes how objects grow and lose their vitality.

As the design process moves from one subtask to the next, successor objects are constructed out of earlier objects. Each new generation has new data fields appended.

Unfortunately, since design often requires cycles, old information cannot disappear. An unacceptable geometry may require a change at the circuit level, say adding an inverter to change a polarity. If design is iterative, then successive transformations of objects must not lose information. This means that objects suitable for one task must contain information relevant to all tasks that may be successors, although much information may be irrelevant to the task at hand. The information may be hidden within the object, but must be passed on correctly, so it remains available when needed.

The objects become big, and no longer have the qualities associated with the object-oriented paradigm. A solution to this problem of overloaded objects is to have super-objects, owned by an object czar. Objects for each user task type are created by projection from the super-object. Updating privileges must be well defined.

This solution does not solve the second class of problems, namely sharing object information when the object configuration differs. Now, to create objects for a user task objects may have to be created from combinations of several and, perhaps, different objects or super-objects.

I show in the next section an EIS example having components and nets connecting. These present different configurations of the same information. In aircraft design the objects serve design tasks as aerodynamics, structures, power, fuel, control, etc., and it is obvious that their configuration is quite different. Even in a simple commercial credit card operation there is the customer as an object for some tasks and the establishments are the objects wanting to be paid in other tasks.

Because of these problems, I present later a proposal that will not try to store to objects directly in persistent form. I prefer a new approach to satisfy the demands of database style sharing and object concepts.

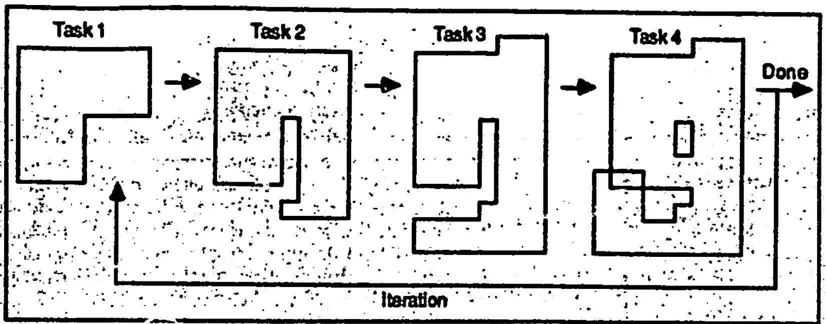

Figure 1. Growth of objects as they try to satisfy multiple design tasks.

Looking at some examples

To clarify the introduction I will take a simple device, a D-type latched flip-flop. At some level of abstraction it is an object; at a lower level it is composed of several gate objects of only three types: AND, INV, NOR, and contacts to the external world. The graphical representation of Figure 2 shows the component objects of the flip-flop at the gate level and the interconnection nets. The components are capitalized and the nets have lowercase labels.

A fully normalized database storing the information describing this circuit requires several relations. I show in Figure 3 the two library relations, which describe the types of gates (Gates) and their connection points (Gate-connects). Many other values will be kept in such a library. The ruling part or key attributes of each relation are placed ahead of a separator symbol :>.

Figure 4 presents a nonredundant representation for the specific circuit using two more relations, one for each gate instance and one giving the net connections for each gate. Other design-specific information can be kept within these relations.

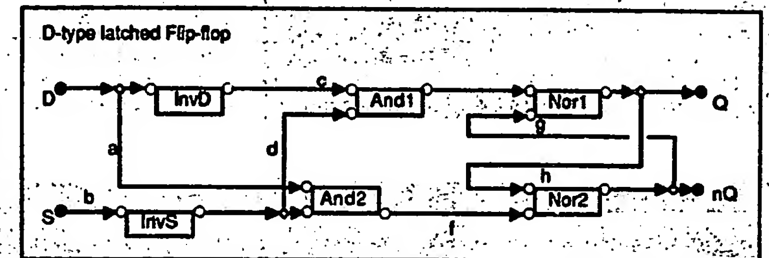

Figure 2. Latched D flip-flop.

Relation Gate-type:)	Gates: Function,	Area,	No-pins,	Power,	Delay,
Inv	$x=/a$	30	2		
And	$x=a\wedge b$	40	3		
Nor	$x=a\vee b$	35	3		
Cntct	a	900	1		

Relation Gate-type,	Gate-connects: C-no :)	C-id,	IO;
Inv	1	a	in
Inv	2	x	out
And	1	a	in
And	2	b	in
And	3	x	out
Nor	1	a	in
Nor	2	b	in
Nor	3	x	out
Cntct	1	a	inout

Figure 3. Relations describing the gates library.

The representation of the design shown is quite complete, but also fairly unclear. I need to create views that place all relevant data into coherent tuples. A view is needed to analyze the components and their loads; another view is needed to look at the nets; and other views will be needed for timing analysis, heat dissipation, etc.

In Figure 5 I extract two views, ComponentsView and NetsView for the database of Figure 4, using also the library relations shown in Figure 3. The ComponentsView is intended to be appropriate for checking components and their sources and sinks. It primarily accesses the Components relation, and joins with its tuples data about the connected components and from the libraries.

The NetView is intended to generate data on the interconnection nets for an application doing checking. The primary relation for the NetView view is the Connections relation, augmented with library information for the connected components.

The number of objects in ComponentsView is equal to the cardinality of the component relation (10), but the augmentation makes the result much larger (30). The eight nets are represented in the primary relation and in the view by 20 tuples, one per connected point (o) or external contact (•). In both views I present the tuples in a logical order.

This view is still awkward, since component entities require multiple tuples. A more reasonable presentation would delete redundancies and add implicit non-first-normal-form bindings by rearranging columns according to the source relations. The Nor1 component shown within Figure 5 would then have a object data structure as shown in Figure 6. I add a column N, which gives the number of connected components. The computation of N using SQL requires Group By and Count operations.

I now describe in Figure 7 this structure in the object-oriented extension of C, namely C++ (reference 6). In C++ structures have to be mappable at compile

Relation Components:

Id:)	Type,	Role,	Position, ...;
D	Cntct	Data in	1/1
S	Cntct	Sense in	3/1
InvD	Inv	Data inverter	1/2
InvS	Inv	Sense inverter	3/2
And1	And	And data	1/3
And2	And	And sense	3/3
Nor1	Nor	Nor to Q	1/4
Nor2	Nor	Nor to nQ	3/4
Q	Cntct	State out	1/5

Relation Connections:

Id,	Pin:)	net;
D	1	a
S	1	b
InvD	1	a
InvD	2	c
InvS	1	b
InvS	2	d
And1	1	c
And1	2	d
And1	3	e
And2	1	a
And2	2	d
And2	3	f
Nor1	1	e
Nor1	2	g
Nor1	3	h
Nor2	1	h
Nor2	2	f
Nor2	3	g
Q	1	h
nQ	1	g

Figure 4. A fully normalized description.

time, so arrays of dynamic extent cannot be embedded: Since it is permissible in C++ to have a dynamic array as the last element of a class by writing an appropriate constructor, it is possible to bring the CCpin array inboard.

Views and objects

Let us now reconsider the similarities between views and object concepts. Both are intended to provide a better level of abstraction to the user, although the database user is seen to manipulate sets of objects in a nonprocedural notation while the objects are manipulated procedurally and iteratively.

The *collection* of tuples of a view is defined by the query that generates the set, and described by the relation-schema associated with the view query. The set of objects is defined by the user-initiated action of generation and described by the prototype. Object-oriented languages may have a ForAll primitive to rapidly generate collections of objects of some type.

The description of the relation has to be available to the relational user, since no implied operations can be kept in the relation-schema. There are proposals to store object-defining procedures in relational schemas, but these have not yet been tested.[5]

Both tuples and objects can be selected based on any of multiple criteria, and interrogated to yield data for processing. View update may be severely restricted because of ambiguities in base relation update. Object update can be restricted by having only limited access functions, but otherwise no constraints are imposed on the programmer, although consistency problems can easily arise among users sharing objects.

Basic, of course, is the difference in persistence. Databases, and hence views over them, persist between program invocations. Objects must be explicitly written to files to gain persistence. Related to persistence is the critical issue to be addressed, namely the multiplicity of views that can be derived from a base relation. Figure 5 showed a view of "components" and a view of "nets" derived from the relations of Figures 3 and 4.

The proposed architectural unit: view-objects

I now propose to combine the concepts of views and objects, as discussed above, into a single concept: view-objects. This proposal is motivated by noting that systems suitable for engineering information problem support require both sharability and complex abstract units, i.e., both views and objects. I use the concept of views to generate a working set of objects corresponding to projections of the entities described in the database.[7] The generators may need to access multiple relations to reconstruct the entities hidden to the relational representation. Components of the architecture are

(1) a set of **base relations** and

(2) a set of **view-object generators.**

In a complete version of this approach I also require the inverse function of (2), namely,

(3) a set of **view-object decomposers** and **archivers.**

The conceptual base relations serve as the persistent database for applications under this architecture. They contain all the data needed to create any specified object type. Conventional database technology should be adequate for development, but eventually performance demands may drive users to specialized databases. I will consider the options for physical organization later.

Concurrent access by long-lived transactions will require a capability to conveniently access prior versions of the data. Where database management systems do not serve that need intrinsically, the database must be configured with additional time-based attributes. As experience is gained, this service will be included in specialized systems being built.

The view-object generator is the new component in this architecture. It performs the following steps:

(a) The view-object generator extracts out of the base database data in relational form as needed—as is now done by a query corresponding to a view. A view tuple or set of view tuples will contain projected data corresponding to an object. A view relation will cover a selected set of objects.

(b) The view-object generator assembles the data into a set of objects. The objects will be made available to the program by attaching them to the predefined object prototype.

```
CREATE VIEW ComponentsView (ID, Pin, IdC :) Type, N, IO, IOC)
        AS SELECT C.Id, C.Pin, CC.Id, CM.Type, T.No-pins, G.IO, GC.IO
        FROM Connections C CC, Components CM, Gate-connects G GC, Gates T,
        WHERE C.Id = CM.Id AND C.Net = CC.Net AND C.Type = T.Gate-type
        AND C.Type = G. Gate-type AND CC.Type = GC.Gate-type;
CREATE VIEW NetsView (Net, Dev, Pd:) IOd)
        AS SELECT C.net, C.Id, C.Pin, GC.IO
        FROM Connections C, Components CM, Gate-connects G GC, Gates T,
        WHERE C.Id = CM.Id AND CM.Type = T.Gate-type;
```

ComponentsView :=

Id,	Pin,	IdC :)	Type,	P,	IO,	IOC;
D	1	InvD	Cntct	1	inout	in
D	1	And2	Cntct	1	inout	in
S	1	Inv1	Cntct	1	inout	in
InvD	1	D	Inv	2	in	inout
InvD	2	And1	Inv	2	out	in
InvS	1	S	Inv	2	in	inout
InvS	2	And1	Inv	2	out	in
InvS	2	And2	Inv	2	out	in
And1	1	InvD	And	3	in	out
And1	2	InvS	And	3	in	in
And1	2	And2	And	3	in	in
And1	3	Nor1	And	3	out	in
And2	1	D	And	3	in	inout
And2	1	InvD	And	3	in	out
And2	2	InvS	And	3	in	out
And2	2	And1	And	3	in	in
And2	3	Nor2	And	3	out	in
Nor1	1	And1	Nor	3	in	out
Nor1	2	Nor2	Nor	3	in	out
Nor1	2	nQ	Nor	3	in	inout
Nor1	3	Nor2	Nor	3	out	in
Nor1	3	Q	Nor	3	out	inout
Nor2	1	Nor1	Nor	3	in	out
Nor2	1	Q	Nor	3	in	inout
Nor2	2	And2	Nor	3	in	out
Nor2	3	Nor1	Nor	3	out	in
Nor2	3	nQ	Nor	3	out	inout
Q	1	Nor1	Cntct	1	inout	out
Q	1	Nor2	Cntct	1	inout	in
nQ	1	Nor2	Cntct	1	inout	out
nQ	1	Nor1	Cntct	1	inout	in

NetsView :=

Net,	Dev,	Pd:)	IOd;
a	D	1	inout
a	InvD	1	in
a	And2	1	in
b	S	1	inout
b	InvS	1	in
c	InvD	2	out
c	And1	1	in
d	InvS	2	out
d	And1	2	in
d	And2	2	in
e	And1	3	out
e	Nor1	1	in
f	And2	3	out
f	Nor2	2	in
g	Nor2	3	out
g	Nor1	2	in
g	nQ	1	inout
h	Nor1	3	out
h	Nor2	1	in
h	Q	3	inout

Figure 5. Component View and Net View.

Id,	Type,	P	Pin,	N,	IO,	IdC,	IOC;
Nor1	Nor	3	1	1	in	And1	out
			2	2	in	Nor2	out
						nQ	inout
			3	2	out	Nor2	in
						Q	inout

Figure 6. Reduced datastructure for a View Object.

```
enum io {IN, OUT, INOUT};
 class Cpin;
class CCpin;
 class ComponentsView
{
        char    Id [8];
        char    Type [6];
        short   P; // Component pin count
        Cpin* Pin;
};
 class Cpin
{
        io      IO;
        short   Q; // Connected component
                   count
        CCpin* C;
};
class CCpin
{
        char*   IdC;
        io      IOC;
};
```

Figure 7. C++ datastructure for Component Objects.

The view-object generator needs information other than the data, i.e., knowledge to perform its functions:

(a1) The query portion identifies the base data.

(a2) The specification of the primary relation identifies the object-entities.

(a3) The object prototype specifies the structure and linkages of the data elements within an object, and the access functions for the object.

Initially the view-object generators will be implemented as code, closely following the translation programs in use now to convert persistent storage representations of engineering data files into representations suitable for specific tools. For experiments a relational database can be used for storage of the base data, but expect that ongoing developments in EIS will provide systems with more appropriate functionality and performance.

The goal is to eventually provide nonprocedural access for object-oriented approaches as well. By formalizing the semantics of the objects required by the tools, and interpreting the object type description, I believe that the view-object generators may be automated. The programmer then only provides the object type declaration and the Where clause defining the desired set of object instances.

With increased storage of data semantics in extended schemas one may advance further. With structural or dependency information about the base relations, automatic generation of the internal structure of an object type may become feasible. Since access to the objects by the tools is still procedural, the performance benefits of automated object generation would be minor. The major benefit is in the control of access and consistency that may be gained.

Justification

I am proposing a major extension to database and object-oriented concepts, with the intent to obtain the joint benefits that each approach provides separately today. The effort must justified by these benefits.

Sharable access to objects. The proposed architecture supports procedural access to objects, as expected by object-oriented programming systems. Access to the base data is automatic, and nonprocedural. This permits base data to be effectively shared, since no single application or combination of applications imposes a structure on the base relations.

Growth of the system. New object types can be defined by adding new view-object generators. As is expected in databases, new data instances, types, and entire relations can be added without affecting other users' programs. Data can be reorganized without changing the object generator code since only the views will be involved. Such flexibility is essential for growth and multiuser access.

Support for a wide variety of representations. When simple tabular results are to be obtained from the database, a view-object generator can easily generate tables for direct inspection and manipulation. A graphics object-generator can project the attributes needed for visual display.

High-performance access from the database. To achieve this goal new database interfaces must be developed that make the benefits of the set-oriented database retrieval concepts available to programming languages. Current database interfaces create a bottleneck when delivering data to programs. A common method for relational systems is to accept a query that specifies a set, but then require repeated invocations that deliver only single values or records at a time. This access mode, because of conventional programming techniques, is clearly inadequate.

The data from the databases is to be inserted into sets of objects at a time. For the object-oriented programmer such a set is best defined as a super-object. A program typically cannot proceed until the set is complete. The object generators need only be invoked once for all the data needed to compose a super-object. An effective view-object generator hence needs an interface at a deeper level into the existing relational functions.

Updating from objects. The programs can freely update the contents of the objects. Some applications require that these changes be made persistent and hence be moved from its object representation to the base database.

To achieve update I envisage a third architectural component, the view-update generator. This component can be invoked at commit-time, when results affecting the objects are to be made persistent. I envisage that such a process will only update from objects that have changed, and only replace values that were changed.

Where views have been constructed using joins, aggregation operators, and the like, automatic view update can be ambiguous. Restrictions on object update are one solution. However, as shown by Keller,[4] such ambiguities can be enumerated and a choice can be made when the view-update is generated. The view update generator can also take advantage of the semantic knowledge available to an advanced view-object generator. An important source of knowledge constraining updates is authorization information.

Cost trade-offs

What costs and benefits will this architecture provide other than what I see as its structural advantage? I will now review areas where performance may be gained versus one of the two underlying approaches—database use or object-oriented programming—alone.

Set-based data access. A single invocation of a view-object generator will instantiate all specified objects. The overhead of programmed access to relations, typically requiring an initial Call giving the query and then iterated Calls to obtain the result piece by piece, is avoided. Also, no sequence of object instance-generating Calls, as seen in object-oriented program-

ming, is needed. Of course, one invocation of the object generator will be a major operation, but only one call should be needed per object type.

The object generator may also perform the so-called macro-generator function, where instances of design objects are generated based on a general prototype and parameters. The source of such information is a library of cell descriptions, permitting, say, the generation of a series of cells making up a register.

The view-object generator will be more complex than either a database retrieval alone or an object prototype. It will provide a very clear definition of the mapping from base data to objects and do so in a localized manner. A partial example was given, using the SQL approach to describe the view in Figure 5 and then using C++ to define the storage structure for the objects in Figure 7.

Binding of retrieved data. The ability of the view-object generator to bind the object will pay off in processing time. No joins are needed at execution to bind relational tuples, and no search operations are needed to assemble tuples belonging to the same object. Since the required information exists at view generation time, no search cost is expended. The only requirement is that the object's internal data structure permit retention of the information.

Task allocation that matches hardware system components. Implicit, but not essential, to my proposal are the complementary concepts of database servers and design workstations. They will be linked by a powerful, but often still, bandwidth-limited communication network. In such an architecture I expect that most of the retrieval and restriction operations will be carried out on the machine serving the database and the object generation and use will occur in the workstation.

Optimization in the file server. The file server can select optimum retrieval strategies and reduce the data volume needed to convey the information. Keeping data in sorted order and removing redundant fields as shown in Figure 6 is straightforward and effective. All operations are specified nonprocedurally and can be arranged and interleaved as needed to handle requests from the users. Concurrency control and version management are tasks best handled in the file server.

Communication minimization. The data volume is reduced in the file server. Communication bandwidth between server and workstation can be used fully for information, rather than for data to be ignored in the workstation.

High performance on workstations. The workstation only needs to assemble the information obtained into the desired object configuration. It is desirable that all objects for an application task can be placed into the real memory of the working processor.[8] The objects now do not contain data fields irrelevant to the process at hand. Since these working objects are now compact, the probability that they will all fit into a modern workstation is increased. Virtual memory management can be invoked if needed, but the capacities of modern machines are such that there should be adequate memory space for working storage of the required objects. However, I can envisage several techniques to exploit having the data independence provided by view generators to improve locality for virtual memory management.

Physical organization of the database

Performance issues are considered critical in EISs, and experiments with relational databases have not given me confidence that adequate performance is easy to obtain. Performance will furthermore be impacted by the extensions, such as version support, that are needed in the engineering environment.[3] Many of these extensions will also be useful for broader objectives, so that there is a motivation to share the development and maintenance costs of relational database systems extensions.

Objects are seen as a means to solve the performance problem, because data is bound in a user-sensible manner. I believe that such binding can indeed be significant in workstations, although my own experiments have not given proof of that assumption when external storage is used.[9] There are obviously many more factors to be considered simultaneously when building comprehensive systems, but storing data in relational tables is often seen as a critical issue.

The fully normalized model of the representation has as its objective the minimization of redundancy and the avoidance of a number of anomalies that can occur. It also supports a very simple storage structure, as seen in our example, that can lead to a large number of relations. Retrieval from such an organization will require a large number of joins, as seen in the example leading to Figure 5.

Updates seem to require less work in a relational database than in our proposal. However, when the database has to obey interrelational constraints, expensive joins must also be executed when the data are updated. For instance, it is necessary to ensure that the components exist and that the connections are valid when a net con-

nection is to be changed.

There is, of course, no requirement for the physical storage structure to mimic literally the logical relational structure. Specifically, information from multiple tuples may be stored in one record. Denormalization has been formally considered, but is not supported by current DBMSs. It is commonly employed in practical relational databases. Preserving correctness in a denormalized database often requires additional transformation and care: records may have repeated fields and at other times some fields may be replicated in multiple records. Sometimes here work is required as well: dependent tuples, as connection points for a component, will automatically be deleted if they were implemented as a repeating group. I expect to profit here from ongoing research on non-first-normal-form databases.

In my architecture, where I expect that the object generator is the primary means for accessing data, complexity issues discouraging use of a non-normalized storage structure are moot. The object generators can be adjusted, probably eventually automatically, to any storage structure chosen. It is likely that the most efficient storage structure will be similar to the dominant object structure.

With such an organization I have provided the two primary objectives favoring the concept of object-oriented programs for computer-aided design:

(1) The concept of a view-object provides the desired clean and compact user interface.

(2) The flexibility of the storage structure can provide the locality needed to achieve high performance in source data access.

I can now provide these objectives for multiple users, and share selected information in a controlled fashion. I gain greatly from the flexibility obtained by interposing the object generator operating on a persistent base storage structure. I am no longer bound to an optimal data storage structure bound to an optimal object format. It is likely that in any design project the data retrieval loads will change over time. The object types that dominate use during initial design phases are not likely to be the object types used during the maintenance phase of the designed devices. A physical storage reorganization is now possible, as long as the view-object generators are adjusted correspondingly.

Replication, the primary tool to improve performance, can be utilized as well. A high-demand subset of the database can be replicated and made available in a performance-effective manner to the object generators. Object update functions can use primary copy token concepts to update all copies consistently and synchronously.

I argue that direct storage of objects, to make them persistent, is not appropriate for large, multi-user engineering information systems. I propose storage using relational concepts, and an interface that generates objects. To test the validity of the concept, initial versions of the interface may be coded directly. In the longer range I look toward knowledge-driven transformations.

Generation of objects from base data brings the advantages of sharing persistent information to an object-oriented program. It also provides a better interface from programs to databases, recognizing that access by programs, large or small, will remain essential for data analysis and complex transactions.

The separation of storage and working representation will also simplify the development of new approaches to engineering design.[10] The object generator can be viewed as a system component utilizing knowledge to select and aggregate data for the information objectives.

Acknowledgments

This paper has benefited from interaction with a VHSIC program-sponsored project to define information models for engineering information systems, although it certainly does not reflect the consensus of the group. Comments by Tony Gadient of the USAF, WPAFB, Arnold Goldstein of MCC, and Professor Alice Parker of USC were constructive and helped focus the issues. Earlier presentations of these concepts at Workshops for Integrated Design and Manufacturing Processes at the Naval Postgraduate School in Monterey, Calif., April 1986, and at the Object-Oriented Database Workshop[1] engendered useful feedback. David Beech, from Hewlett-Packard, visiting the Stanford Center for Integrated Systems, provided motivation to consider the object-oriented approach and gave constructive criticism to earlier versions of this paper. He also provided the C++ example for Figure 7. Support for this research was provided in part by DARPA contract N39-84-C-0211 for Knowledge Based Management Systems and in part by the Air Force through a contract with UES F33615-82-C-1716.

References

I can cite here only a few of the many publications that have provided concepts leading to this paper. I include some recent global references to provide access to other work. Further references can be obtained from me.

1. K. Dittrich and U. Dayal (eds.), *Proc. 1986 Int'l Workshop on Object-Oriented Database Systems*, IEEE-CS order no. 734, Sept. 1986.
2. F. Lochovsky (ed.), *Object-oriented Systems*, special issue of the *IEEE-CS Database Eng. Bull.*, Dec. 1985; reprinted in *Database Engineering 1985*, IEEE-CS order no. 758, 1986.
3. R. H. Katz and T. J. Lehman, "Database Support for Versions and Alternatives for Large Files," *IEEE TSE*, Vol. SE10, No. 2, Mar. 1984, pp. 191-200.
4. A. M. Keller, "Choosing a View Update Translator by Dialog at View Definition Time," *Computer*, Jan. 1986.
5. M. Stonebraker and L. Rowe, *The Design of POSTGRES*, tech. report UC Berkeley, Nov. 1985.
6. B. Stroustrup, *The C++ Programming Language*, Addison-Wesley, Reading, Mass., 1986.
7. H. M. Carter, "Computer-Aided Design of Integrated Circuits," *Computer*, Vol. 19, No. 4, Apr. 1986, pp. 19-36.
8. L. Holland et al., "Engineering Support System Software," *Micro*, Vol. 5, No. 5, Oct. 1985, pp. 17-21.
9. M. W. Wilkins et al., "Relational and Entity-Relationship Model Databases and Specialized Design Files in VLSI Design," *Proc. ACM-DA 22*, 1985, pp. 410-516.
10. D. J. Hartzband and F. J. Maryanski, "Enhancing Knowledge Representation in Engineering Databases," *Computer*, Vol. 18, No. 9, Sept. 1985, pp. 39-48.

Gio Wiederhold is an associate professor of medicine and computer science (research) at Stanford University. A member of the Computer Systems Laboratory and the Center for Integrated Systems, he is active in advanced database management and its applications to medicine and planning. On the editorial board of IEEE *Computer*, ACM's *Transactions on Databases*, IEEE *Expert*, and Springer-Verlag's *MD Computing*, Wiederhold has been involved in over 80 publications in computing and medicine.

Wiederhold received Lyceum and undergraduate training in aeronautical engineering in Holland. He held positions as programmer, manager, systems architect, and laboratory director. After 15 years in industry he returned to academia, obtaining in 1976 a PhD in medical information science from the University of California.

Readers may write to the author at the Department of Computer Science, Stanford University, Stanford, CA 94305.